REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
26-04-2017	Final Technical	31-07-2013 to 30-07-2016

4. TITLE AND SUBTITLE
Integrating Security in real-Time Embedded Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER
N00014-13-1-0707

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Mohan, Sibin; Bobba, Rakesh B.; Pellizzoni, Rodolfo; Nicol, David M.;

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Board of Trustees of the University of Illinois at Urbana Champaign
1901 South First Street, Suite A
Champaign, IL 61820

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
875 North Randolf Street
Arlington VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Our main focus is to obtain an understanding of how security could be effectively integrated into real-time systems. Currently, there does not exist a comprehensive theoretical framework for the integration of security in embedded real-time systems. Traditionally, real-time systems are modeled as a set of a periodic tasks with timing constraints that are then scheduled on a collection of resources (e.g., the processor, memory, bus, etc.). A key insight that we believe will be useful is in identifying this resource management nature of RTS and the need for security policies to adhere to the strict guidelines imposed as a result.

15. SUBJECT TERMS
Real-Time Embedded Systems, security, integration

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Sibin Mohan
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	SAR	12	217-300-3037

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATE COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33315-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report. e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organization, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.

Final Technical Report – N00014-13-1-0707

Integrating Security in Real-Time Embedded Systems

April 26, 2017

Complete List of PIs and their Affiliations:
Sibin Mohan, Information Trust Institute, University of Illinois at Urbana-Champaign
Rakesh Bobba, School of Electrical Engineering and Computer Science, Oregon State University
Rodolfo Pellizzoni, Dept. of Electrical and Computer Engineering, University of Waterloo
David Nicol, Information Trust Institute, University of Illinois at Urbana-Champaign

Technical Contact:
Sibin Mohan
Information Trust Institute
University of Illinois
1308 W. Main St.
Urbana, IL 61801.
Phone: 217-300-3037
Email: sibin@illinois.edu

1 Introduction

Our main focus is to obtain an understanding of how security could be effectively integrated into real-time systems. Robust systems must be able to both prevent expected attack vectors from succeeding, as well as comprehensively monitor themselves and detect intrusion events. Currently, there *does not exist* a comprehensive theoretical framework for the integration of security in embedded real-time systems.

Traditionally, real-time systems are modeled as a set of a periodic tasks with timing constraints that are then scheduled on a collection of resources (*e.g.*, the processor, memory, bus, *etc.*). A key insight that we believe will be useful is in identifying this resource management nature of RTS and the need for security policies to adhere to the strict guidelines imposed as a result. Hence, any security related mechanisms must work within the imposed restrictions of timeliness, determinism, *etc.* On the other hand, these properties of RTS actually make it easier to model such systems and perform a rigorous analysis of any solutions that will be developed.

1.1 Specific Problem and Tangible Results

The goals of this effort are the development of a new, generalized, *framework for the integration of security and real-time systems* that will involve a combination of research along the following axes:

1. Understanding and classification of current and emerging *threat landscapes* for such systems – what types of vulnerabilities exist, what attacks are possible, what security properties are essential.

2. Security policies and system models that bring the seemingly diverse areas of real-time systems and security together to develop a framework for resource management algorithms in real-time systems.

3. Development of *security mechanisms* that leverage real-time resource managers that will *(a)* prevent attacks from being successful and/or *(b)* detect any intrusions/attacks once they occur and *(c)* keep the overall system safe in the event of an attack.

4. Analysis and evaluation of the effectiveness of the proposed mechanisms, both in terms of their capability to meet real-time requirements, as well as their ability to improve the security of the system. We will follow an experimental methodology based on the development of suitable case studies and demonstration platforms.

1.2 Approach

We are working on understanding how the resource management nature of such systems affects the process of integrating security. To this effect, we analyze existing systems to gain and understanding of *(a)* what security problems can manifest themselves and *(b)* how they can be fixed in the context of the various resource managers in the system. We will present some initial work along these lines in Section 2.

1.3 Research Plan and Milestones

We proposed a three year plan of work for our research and continue to make progress along the same lines. In the **first** year, we focused on what we call a "*vertical slice*" of the problem to show the feasibility of the approach. We modeled a set of real-time tasks with security relationships ordered as as lattice of security levels. In the **second** year, we significantly expanded the scope of the work to improve the analysis, developed a more general model (what we called a "vendor-based model") to describe security relationships between tasks, analyzed the effects of preemption and resource (cache) partitioning and also started to develop ideas on how to carry out an attack on real-time systems. In the **third** year (and beyond), we expanded our work in both security integration and attack mechanisms, and worked on demonstrations and evaluations in hardware.

Year I: In the first year, we looked at a specific range of issues in each of the domains: *viz.*, security vulnerabilities, real-time architectures, scheduling and resource management algorithms and mitigation mechanisms, all in the scope of a particular application domain. For our application domain, we focused on

unmanned aerial vehicles (UAVs)[1] Sections 2 and 2.3 provide details on our work on these topics.

In the first year, we focused on security attacks that threaten the confidentiality and integrity of real-time systems. Our research was aimed at mitigating such problem by creative scheduling (and broader resource management) algorithms. While our initial work is focused on single-core systems, we plan to expand this to multicore systems – they bring up interesting problems of their own due to the co-location of real-time tasks. We also plan to generalize our application model (from Sections 2.3 and 2.3.2) that can then be used by us (and the real-time community) for research on real-time security.

Year II: We expanded the work from the first year by generalizing the model to describe the security relationships between real-time tasks. We named it the *vendor-based model*. We then developed a close-to-optimal algorithm to determine the overhead for the security mechanisms developed in year I. We also analyzed the effects of preemption in such scenarios. Another related topic that we explored was how a partitioning of the underlying shared resources (*e.g.*, caches) help reduce the effects of attacks that we explored in year I (leakage of information through the cache).

Finally, we started working on demonstrating actual attacks that can be used to leak information. This process requires two steps: *(a)* an understanding of the exact schedule of tasks that are executing and *(b)* using this information to launch a cache timing attack on a specific task.

Year III: In the third year, we further expanded our work in both security and attack mechanisms from the second year. We developed a scheme to integrate security policies in *legacy* systems using *opportunistic execution* [3]. A metric, by means of *achievable periodic monitoring*, was proposed to measure security of the system in such work. We also develop a schedule randomization protocol, *TaskShuffler* [8], to reduce periodicity of the schedules in hard real-time systems that will make it harder for attacker to extract system behavior. On the other hand, the developed schedule-based attack from the second year is further refined to tolerate variations in system attributes, making the attack more applicable to realistic platforms [1,2].

1.4 Advantages of the Approach

The main advantages of this approach are:

1. Introduction of security in RTS at a holistic level;
2. Makes the design of RTS inherently more secure;
3. Will not require hardware changes, at least initially;
4. Better understanding of the security challenges in the field of Real-Time Systems.

2 Current State and Results

Publications:

1. "*Real-time Security through Scheduler Constraints*" published in Euromicro Conference on Real-Time Systems (ECRTS), Madrid, July 2014.
2. "*A Generalized Model from Preventing Information Leakage in Hard Real-Time Systems.*" published in Real-Time and Embedded Systems and Applications (RTAS), Seattle, April 2015.
3. "*Integrating Security Constraints into Fixed Priority Real-Time Schedulers*" published in Real-Time Systems Journal, 2016.
4. "*TaskShuffler: A Schedule Randomization Protocol for Obfuscation Against Timing Inference Attacks in Real-Time Systems*" published in 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2016.

[1] These ideas can also be applied to unmanned underwater vehicles (UUVs). In this work we focus on UAVs and will leave the demonstration on UUVs for future work.

5. *"ScheduLeak: An Algorithm for Reconstructing Task Schedules in Fixed-priority Hard Real-time Systems"* presented in 1st Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS), November 2016.
6. *"Exploring Opportunistic Execution for Integrating Security into Legacy Hard Real-Time Systems"* in IEEE Real-Time Systems Symposium (RTSS), November 2016.

We now present detailed information about what we have achieved in the past three years.

2.1 Integrating Security into Real-Time Scheduling

We identified one security problem – that of confidentiality when a system has shared resources, *e.g.*, caches, buses, *etc.*Our initial focus is to identify how this problem can be reconciled with the traditional real-time scheduling requirements on single core processors.

2.1.1 Information Leakage Prevention

It has been shown that the use of shared resources makes it possible for information to be leaked between tasks without the use of explicit communication [4, 6]. The issue of covert timing channels between tasks of different security levels in the RM scheduler was previously considered [7]. We, in contrast, focus on information leakage due to the sharing of resources[2] such as the cache, DRAMs, I/O bus, *etc.*

In our Year I work, we proposed the following: *casting security-related requirements in the form of real-time scheduling constraints*. This enabled us to directly reason about the effects of integrating security into RTS (*e.g.*, effects on schedulability). The problems related to information leakage between real-time tasks with different security levels was an example of a security issue that must be solved by these techniques. We considered a uniprocessor system following the Liu and Layland task model [5] that contains a set of sporadic tasks scheduled based on the Rate Monotone (RM) policy. We are concerned with the leakage of information between tasks of different criticality. While we started by considering a standard security model consisting of a lattice of security levels, we found that in some cases this model can be limited in scope. Hence, we *developed a new, generic model* (in **Year II**) that can capture the exact security relationships between every pair of tasks in the system; we discuss this *vendor-oriented* model in more details in Section 2.3.2.

More often than not, every time there is a switch between tasks belonging to different security levels there is a possibility of information leakage through shared resources such as the cache. We believe that through the use of intelligent scheduling mechanisms it is possible to integrate security at the design phase of RTS and reduce this potential for information leakage via shared resources. A cost must be paid for each shared resource in the system to prevent this leakage of information. In our work, we discuss various methods of integrating such a penalty (and associated constraints) into scheduling policies for real-time systems and derive analysis bounds for the same.

We developed two mechanisms to prevent information leakage between two tasks. The first method consists in the use of a synthetic 'flush task' (FT) that resets the state of any resource that might be used to leak information. For example, in the case of a cache, we can simply evict all cache lines. Note this incurs some overhead, both because dirty cache lines needs to be written back to main memory upon executing the FT, and because useful cache line might be evicted, forcing the next executing task to reload them in cache.

The second mechanism consists in partitioning the resource: if we assign to two tasks disjoint sections (partitions) of the shared resource, then there is no way for the tasks to communicate through said shared resource. However, this reduces the amount of resource available to each task, thus potentially incurring a timing penalty. For example, cache can be partitioned using techniques at either the hardware or software level, but reducing the amount of cache available to any given task might increase the number of capacity misses and thus increase the execution time of the task.

[2]Other than the processor core.

In the first year of the proposal, we focused on the simpler model considering an ordered set of security levels. Based on this model, we proposed a set of scheduling constraints that can be used to guarantee the security properties using a flush mechanism. We described the effect of such constraints over a variety of real-time scheduling algorithms, and devised a schedulability analysis for the simplest case (non-preemptive fixed-priority scheduling).

In Year II, we significantly expanded the scope of the work by achieving the following major contributions:

1. We developed a close-to-optimal algorithm to determine the number of flushes required to guarantee the security requirements for a given task under analysis under the vendor-oriented model.

2. We extended such algorithm to take into account the effects of partitioning.

3. Based on the previous two points, we derived a schedulability analysis to determine whether a set of real-time tasks can be scheduled while guaranteeing all security requirements based on a combined flushing and partitioning scheme.

4. Our investigation reveals that whether a task is executed preemptively or not has a large impact on the overhead of the devised security mechanisms. Hence, we developed an algorithm to optimally determine whether each task should be executed preemptively or not.

5. We created a design-exploration methodology to allow the designer to search for an optimal system configuration. The main idea behind our design exploration technique is that different tasks are more or less susceptible to the effects of partitioning, *i.e.*, tasks with small working sets can be assigned to small partitions, while tasks with large working sets need the entire resource. This allows us to reduce the overhead of flushing by assigning less-sensitive tasks to separate partitions.

6. To study the effectiveness of our solutions, we are currently working on demonstrating an attack on a typical unmanned vehicle platform – we intend to show how an attacker can gauge the exact schedule of the system and then use it to launch a cache-based side-channel attack. Section 2.2 provides more information on this topic.

2.1.2 Integrating Monitoring and Detection Mechanisms in Legacy RTS

Given the increasing risk of cyber attacks, it is essential to have a layered defense and integrated resilience against such attacks into the design of RTS. However, any security mechanisms have to *co-exist* with real-time tasks in the system and have to operate without impacting the timing and safety constraints of the control logic. Besides, the embedded nature of these systems limits the availability of computational power (*e.g.*, memory or processor) required for resource-extensive monitoring mechanisms. This creates an apparent tension between security requirements (*e.g.*, having enough cycles for effective monitoring and detection) and the timing and safety requirements. For example, a critical parameter is to determine how often and how long should a monitoring and intrusion detection task execution to be effective but not interfere with real-time control or other safety-critical tasks. While this tension could potentially be addressed for newer systems at design time, this is especially challenging for retrofitting *legacy* systems where the control tasks are already in place and perhaps *cannot be modified*. Any hardware and/or software-level modifications to those legacy system parameters is costly since it will go through several verification and validation steps and may increase system downtime.

Given the tension between security and timing requirements, while integrating security mechanisms into a practical system, finding the *frequency of execution* of the monitoring tasks is an important design parameter that trades security requirements with timing constraints. If the interval between consecutive monitoring events is too large, the adversary may harm the system (and remain undetected) between two invocations of the security task. In contrast, if the security tasks are executed very frequently then it may

impact the schedulability of the real-time tasks. Herein lies an important trade-off between *monitoring frequency and schedulability*.

To integrate security tasks into legacy RTS, in preliminary work [3] we address the problem of determining the frequency of execution (*e.g.*, periods or inter-monitoring interval) of the security tasks. Our approach to integrate security without perturbing real-time scheduling order is to execute security tasks at a *lower priority* tasks than real-time tasks. We refer this scheme as *opportunistic execution* since the security tasks are only allowed to execute opportunistically only during *slack times* when no other real-time tasks are running.

We propose to measure the security of the system by means of the *achievable periodic monitoring*. Let T_i be the period of the security task that needs to be determined. Since our goal is to minimize the perturbation between the achievable (*i.e.*, unknown) period T_i and the desired period T_i^{des}, we define the following metric: $\eta_i = \frac{T_i^{des}}{T_i}$ that denotes the *tightness* of the frequency of periodic monitoring for the security task τ_i. Thus $\eta = \sum_{\tau_i \in \Gamma_S} \omega_i \eta_i$ denotes the *cumulative tightness* of the achievable periodic monitoring for a set of security tasks Γ_S where ω_i is the designer provided weighing factor that may reflect criticality or severity of the security tasks. This monitoring frequency metric provides one way to trade-off security with schedulability – if the interval between consecutive monitoring events is too large, an adversary may remain undetected and harm the system between two invocations of the security task, on the other hand, a very frequent execution of security tasks may negatively impact the schedulability of the real-time tasks. Hence, to find the desired η of the target system we formulate a constraint optimization problem and also developed a polynomial-time solution that allows us to execute security routines with a frequency closer to the desired values while respecting the temporal constraints of the other real-time tasks.

2.1.3 Schedule Randomization Protocol

One way to protect a system from certain attacks (*e.g.*, the schedule-based side-channel attack) is to *randomize the task schedule* to reduce the deterministic nature of periodic real-time applications. By randomizing the task schedules we can enforce non-determinism since every hyper-period will show different order (and timing) of execution for the tasks. Unlike traditional systems, randomizing task schedules in RTS is not straightforward since it leads to priority inversions that, in turn, may cause missed deadlines – hence, putting the safety of the system at risk.

Hence, we proposed *TaskShuffler* [8], a randomization protocol for fixed-priority scheduling algorithm, to achieve such randomness in task schedule. For instance, by picking a random task instead of the one with the highest-priority at each scheduling point, subject to the deadline constraints. The degree of randomness is flexible in *TaskShuffler*. Based on the system's needs, *TaskShuffler* implements the following randomization schemes:

- *Randomization (Task Only)*: This is the most basic form of randomization in contrast to other schemes introduced below. We randomly pick a task to execute whenever a task arrives or finishes its job, *i.e.*, at the scheduling points. The effectiveness against the schedule-based side-channel attack is limited since the busy intervals in this scheme remains the same.

- *Randomization with Idle Time Scheduling*: In addition to the randomness provided in the basic scheme, we include the *idle task* (*e.g.*, the dummy task executed by an RTOS when other real-time tasks are not running) at each scheduling point. It eliminates the periodicity of busy intervals (from hyper-period's point of view). This scheme makes it harder to produce effective results from the schedule-based side-channel attack.

- *Randomization with Idle Time Scheduling and Fine-grained Switching*: To push the randomization to an extreme, one could choose to randomize the schedule every tick. That is, the scheduler will randomly pick a task to execute, subject to the deadline constraints, in every tick interrupt. This way, we gain the most randomness for the schedule. However, it greatly increases the overheads and thus may not be applicable for all use cases.

2.2 Attacks and Analysis

One facet of our research is to study the mechanisms that an attacker can use to enter the system and the information that can be gained as a result. In this section we enumerate our ongoing work on *attack mechanisms* for real-time systems.

While examining various side channel attacks and the information each attack can reveal in real-time systems, we identified that an adversary can potentially reconstruct the schedule of a hard real-time system by observing the active states of the scheduler. A leakage of scheduling behavior of the system gives attackers sufficient information to pinpoint the start point of any selected task and launch an attack with an increased precision (*e.g.*, cache timing attack) with minimum footprint. Since this study focuses on the leakage inside of the system, we assume that the attacker has already gained control of one or more user tasks as well as some amount of task information such as task periods and execution times. Adversaries in this scenario can gain access to the system due to vulnerabilities in either the system integrator or untrusted vendors.

We developed a sophisticated algorithm, *ScheduLeak* [1, 2], to reconstruct the scheduling based on the gathered active state data – it is being implemented on both, Zedboard (a ARM Cortex-A9 development platform running FreeRTOS) and a simulation tool that we specifically built for verifying and analyzing experiment results. Furthermore, the cache timing attack that follows the reconstruction of the scheduling is also implemented on the Zedboard.

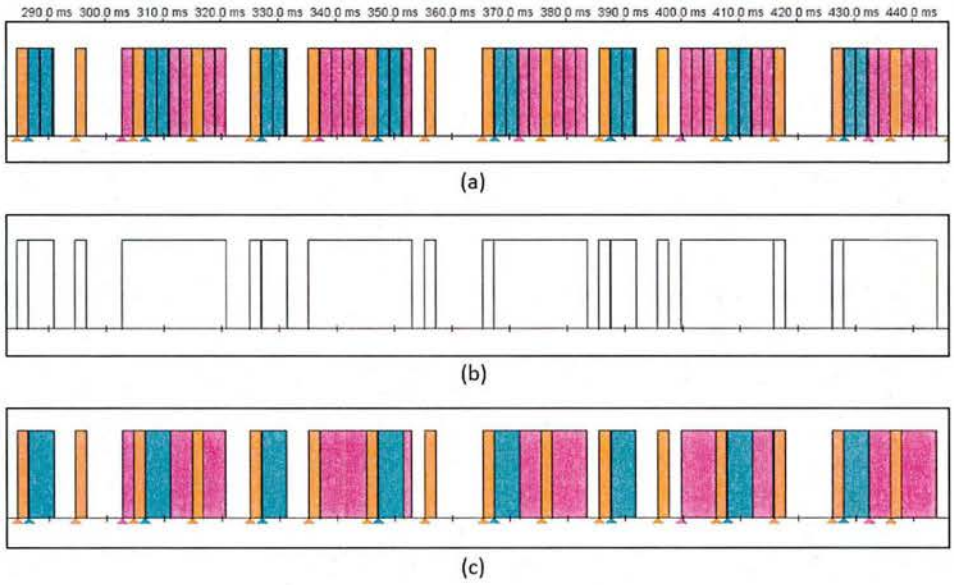

Fig. 1: Experiment Result of The Scheduling Reconstruction. *(a)* Ground truth of the scheduling. *(b)* Busy intervals captured by idle task. *(c)* The result of scheduling reconstruction.

2.2.1 Schedule Reconstruction

To capture the state of the system schedule, we propose to use an *observer task*. The *observer task* can be either the lowest priority task owned by the attacker or the idle task that is injected an "observing function". Being the lowest priority task in a preemptive real-time system makes it possible to measure the active time of the entire system.

However, measuring the active time doesn't provide us with enough direct information about the schedule. In fact, active time intervals are more like pieces of separate chunks of *busy intervals* that are composed of unknown number of jobs from each task. Thus, in order to reconstruct the schedule, we develop an al-

gorithm to analyze each busy interval as follows:

1. Analyze each busy interval individually and infer possible task combinations.
2. Calculate the arrival time window of each task based on the inferred task combinations in each busy interval.
3. Reconstruct the scheduling for each busy interval with the calculated arrival time window.

Step 1 focuses on the problem of finding the quantity of each task constituting a busy interval. Note that every busy interval is analyzed independently in this step and some busy intervals may result in multiple inferences. Step 2 infers the possible arrival time window of each task in each busy interval. For one task, the advanced arrival time window is calculated by intersecting all the possible windows of the task in every busy interval. The arrival time windows are then used by a compact RM scheduling simulator specific for reconstructing the scheduling of any selected busy interval in step 3.

Figure 1 presents the result of the reconstruction on the Zedboard. Part *(a)* shows the ground truth (the actual schedule) that we want to reveal from the system. Part *(b)* depicts the busy intervals captured by the hijacked idle task. These busy intervals are processed by an analysis algorithm, result from which are presented in part *(c)* . With the comparison between ground truth and the result of reconstruction, it shows that the proposed attack scheme can successfully restore the scheduling by merely observing the active state of the system.

In the second year, the algorithm works under the assumption that the task execution times are fixed to the worst case execution times. This limits the developed attack scheme to the systems without uncertainty. In the third year, we further relax this assumption and refine the algorithm to make it tolerate variations in task execution times. This makes the attack applicable to more realistic use cases.

The results gained from this process will be used to launch a cache timing attack on a specific task as explained in the following section.

Note: The above work on schedule reconstruction was carried out in collaboration with Prof. Negar Kiyavash's group at UIUC.

2.2.2 Cache Timing Attack

A cache timing attack is a side-channel attack that manipulates an indirect (storage) leakage channel to steal certain information from a system. The attack exploits the leakage channel through the cache memory that makes it possible to infer the cache memory usage of a specific task.

In our attack scheme, we consider compromising the highest priority task. In contrast to the idle task or the lowest priority task, a task with the highest priority has the shortest period (rate-monotonic scheduling) that gives it privileges to be scheduled with a higher frequency and finish a complete job without being interrupted. These characteristics are beneficial for the attacks that require computation of algorithm in a continuous time, *e.g.*, cache timing attack.

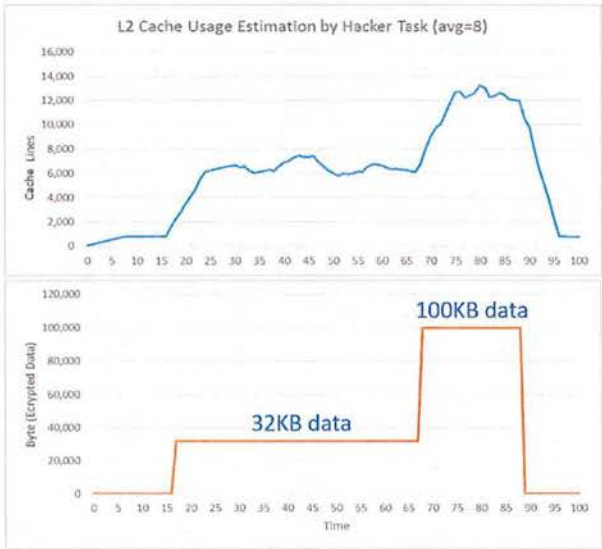

Fig. 2: Experiment Result of The Cache Timing Attack

In our implementation, the attack is triggered once we detect the execution of a "target" task – based on the scheduling analysis from Section 2.2.1. The compromised task (with highest priority) launches the cache timing attack right at the end of the execution of the busy interval that includes the target task. It measures the cache memory usage for the target

task. By replicating this process multiple times, we can reasonably infer the memory usage behavior of the target task.

Figure 2 shows the experimental results of the attack targeting a task that switches between two memory consumption/operational modes: *(i)* processing 32KB images and *(ii)* processing 100KB images. From these results, the cache usage behavior of the target task can be successfully inferred and the two operation modes are distinguishable.

2.3 Demonstration Platform

In general, a big challenge for real-time system researchers, in spite of innovating new techniques for common system issues and supportive mathematical proofs, is to seek a realistic hardware platform to demonstrate and implement their solutions in practice. To address this challenge, we have decided to leverage the Hexacopter Avionics Case Study developed by the Real-Time Embedded Systems Lab (RESL) at the University of Waterloo (UW). In particular, we use this case study to:

- derive implementation parameter (*e.g.*, flush time, partitioning overhead) required to drive our theoretical research;
- test the implementability of devised security mechanisms;
- evaluate mechanisms on a realistic case study of an unmanned vehicle.

The existing Hexacopter case study only consisted of software running on two Beaglebone boards: one running a hardware-in-the-loop (HIL) module (as a simulated model of the real Plant), and another board (Electronic Control Unit, ECU) running an Autopilot application. The HIL module is connected to a real hexacopter with 3-degrees of freedom to retrieve roll, pitch and yaw information; the HIL itself simulates the position of the vehicle. In this way, the platform can be realistically tested without the need to fly the UAV outside[3].

Our demonstrator builds on top of the existing Hexacopter ECU. In abstract, sensor data (both real and simulated) are fed from the HIL to the ECU over a bidirectional serial communication port. Part of the sensor data, such as GPS, which are forced to be simulated (due to indoor constraints), are generated by the simulated model of the plant; the rest are produced by the Hexacopter inertial navigation sensors. After making done some computations on the received sensors data, the actuation data is sent back into the simulated plant, again through the same serial communication port. The main tasks in charge of the above functionalities communicates through global data structures which are synchronized by control variables. The controller/actuator task that computes the control action and sends actuation data to the HIL is periodic with a frequency of 50Hz.

To properly demonstrate and test our devised security mechanisms on the Hexacopter platform, we mainly worked in two directions. First, we implemented a suitable hardware/software platform and ported the Autopilot application to the new platform; second, we created a more complex case study by including additional sensing and communication tasks in the ECU.

2.3.1 Platform Implementation

To allow quick prototyping and testing of the devised mechanisms, we decided to migrate the HW/SW platform to a multicore FPGA-based ARM platform running the FreeRTOS operating system. The configurability of the FPGA-based platform allows us to easily test hardware-based mechanisms. At the same time, the FreeRTOS kernel is lightweight and simple, making a perfect candidate to quickly implement and test various security-aware scheduling mechanisms.

As a first step, we performed the actual HW port, ensuring that the HIL could be connected to the new ECU hardware, a Xilinx Zynq FPGA board that can implement either a dual core ARM A9 processor or a

[3]Note that UAV regulation is stricter in Canada compared to the US: Canada's federal transport administration requires written permission for every experiment performed in the airspace.

2-8 soft core systems (Xilinx Microblaze). We also configured FreeRTOS to run on the Zynq platform. Then, we worked on porting the software application and implementing our devised security mechanisms:

1. Autopilot port: we ported the autopilot application to FreeRTOS running on a single core ARM system. The previous autopilot version was running on Linux, which is unsuitable to conduct experiments on hard real-time scheduling. Due to differences in library support and execution semantics, the code had to be significantly updated.

2. Platform characterization and evaluation: we characterized both the hardware platform and the application by extracting relevant metrics, including scheduling overhead, flush time, worst-case execution times based on resource partitioning, etc., to better characterize our scheduling analysis.

3. Flush mechanism implementation: we extended FreeRTOS to implement the flushing mechanism for both level 1 and level 2 cache. Our implementation leverages hardware support to efficiently perform the cache flush, and is able to check whether a flush is required in constant time in the number of tasks.

4. Partitioning implementation: we implemented a partitioning for level 2 cache. Our system supports cache-way partitioning, where each task can be assigned to one or more of the 8 available ways in level 2 cache.

5. Scheduling implementation: we implemented the devised scheduling algorithm in FreeRTOS. In particular, the OS has been extended to selectively allow tasks to execute either preemptively or non-preemptively. Also, the scheduler has been modified to check at run-time whether a FT execution is required before switching to a new task.

Note: While most of the above was implemented at the University of Waterloo, we are currently developing some aspects of the platform at the University of Illinois. So far, we have been able to port FreeRTOS on to an dual core ARM zedboard system and also set up the various drivers and software tasks. Our initial objective is to demonstrate a leakage-based attack (Section 2.2) on a realistic platform and the show the efficacy of our proposed solutions.

2.3.2 Avionics Case Study and Security Model

The original Hexacopter case study comprised only the Autopilot application. To build a more involved demonstrator, we added other applications to the platform to perform additional sensing and communication tasks.

Figure 3 shows the main tasks comprising our new case study, as well as their communication patterns. The sensor, control law and actuator tasks comprise the existing Autopilot application. We assume that the UAV is employed to capture reconnaissance video through an integrated camera. Images are captured from the camera by the corresponding driver, encoded in a compressed format, and then encrypted before being transmitted to the base station. We further assume that the Autopilot and Camera subsystems have been coded by different sub vendors. Finally, the system integrator connects the two subsystems by means of a mission planner tasks, which determines navigational way-points, and a network subsystem that exchanges information with a base station through wireless communication. Some tasks must be protected (*e.g.*, because of their security classifications, or because they contain proprietary algorithms) and should not leak information to tasks of other vendors. This vendor-oriented security model creates a non-trivial security relation between the various tasks, which makes enforcing security requirements challenging. **Note:** This above model was developed in collaboration with Stanley Bak, a researcher from Air Force Research Labs, Rome, NY.

In particular, we were able to demonstrate that the discussed vendor-oriented security constraints cannot be modelled by a standard multi-level security model. This is because standard models consider a lattice (i.e., a partial order) of security levels, which implies that security relations must be transitive: if a task τ_1 must be protected from τ_2 and τ_2 must be protected from τ_3, then τ_1 should be protected from τ_3 as

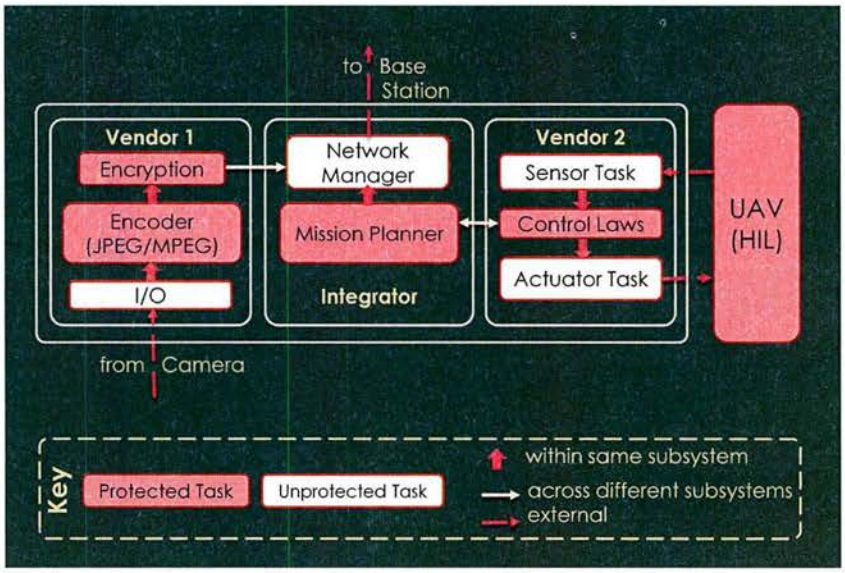

Fig. 3: Avionics Case Study: Tasks and Protection Assignments

well. However, this is not the case in our discussed case study: for example, the Encryption task must be protected from Control Laws, and Control Laws must be protected from Camera Driver, but Encryption does not need to be protected from Camera Driver, since they belong to the same sub vendor. Hence, to properly capture the requirements in our case study, we developed a new generic model consisting of a matrix of security requirements (no-leakage) between any two tasks. The model has then been used as the basis of our theoretical analysis discussed in Section 2.1. Furthermore, the FreeRTOS implementation described in Section 2.3.1 directly implements the discussed model by incorporating the no-leakage relation in the descriptor of each real-time task.

In Year I, we ported the hexacopter demonstrator to the described platform, implemented the avionics case study, and realized the flush mechanism. In Year II, we devised the vendor-oriented model, implemented the partitioning scheme, performed platform characterization and testing, and implemented the devised schemes in the FreeRTOS scheduler.

[1] Chien-Ying Chen, Rakesh B Bobba, and Sibin Mohan. Schedule-based side-channel attack in fixed-priority real-time systems. Technical report, University of Illinois, `http://hdl.handle.net/2142/88344`, 2015. [Online].

[2] Chien-Ying Chen, Amiremad Ghassami, Sibin Mohan, Negar Kiyavash, Rakesh B. Bobba, and Rodolfo Pellizzoni. Scheduleak: An algorithm for reconstructing task schedules in fixed-priority hard real-time systems. In *IEEE CERTS*, pages 39–46, 2016.

[3] Monowar Hasan, Sibin Mohan, Rakesh B Bobba, and Rodolfo Pellizzoni. Exploring opportunistic execution for integrating security into legacy hard real-time systems. In *IEEE RTSS*, pages 123–134, 2016.

[4] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[6] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[7] Joon Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems. In *Information Assurance Workshop, 2006 IEEE*, pages 361–368, 2006.

[8] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *IEEE RTAS*, pages 1–12, 2016.